Linear-Time Sorting Algorithms

Sorting So Far

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

Sorting So Far

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn’t sort in place

Sorting So Far

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place

Sorting So Far

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - $O(n \lg n)$ average case
  - Fast in practice
  - $O(n^2)$ worst case
    - Naïve implementation: worst case on sorted input

Comparison sort

- Comparison sort:
  - Insertion sort, $O(n^2)$, upper bound in worst case
  - Merge sort, $O(n\lg n)$, upper bound in worst case
  - Heapsort, $O(n\lg n)$, upper bound in worst case
  - Quicksort, $O(n\lg n)$, in average case
- Question:
  - what is the lower bounds for any comparison sorting: i.e., at least how many comparisons needed in worst case?
  - It turns out: Lower bound in worst case: $\Omega(n\lg n)$, how to prove?
  - Merge and Heapsort are asymptotically optimal comparison sorts.

How Fast Can We Sort?

- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements

Sorting In Linear Time

- Counting sort
  - No comparisons between elements!
  - ***But***…depends on assumption about the numbers being sorted
    - We assume numbers are in the range *1*.. k
  - The algorithm:
    - Input: A[1..n], where A[j] $\in$ {1, 2, 3, …, k}
    - Output: B[1..n], sorted (notice: not sorting in place)
    - Also: Array C[1..k] for auxiliary storage

Counting Sort

```
CountingSort(A, B, k)
        for i=1 to k
                C[i]= 0;
        for j=1 to n
                C[A[j]] += 1;
        for i=2 to k
                C[i] = C[i] + C[i-1];
        for j=n downto 1
                B[C[A[j]]] = A[j];
                C[A[j]] -= 1;
```

Work through example: A={4 1 3 4 3}, k = 4

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What will be the running time?

Counting Sort

- Total time: $O(n + k)$
  - Usually, $k = O(n)$
  - Thus counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$!
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is *stable*

Counting Sort

- Cool! *Why don't we always use counting sort?*
- Because it depends on range *k* of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, *k* too large (2^{32} = 4,294,967,296)